

Outstanding User Interfaces with Shiny

David Granjon

2020-05-11

Contents

Prerequisites	7
Disclaimer	7
Is this book for me?	7
Related content	7
1 Introduction	9
Survival Kit	13
2 HTML	15
2.1 HTML Basics	15
2.2 Tag attributes	16
2.3 HTML page: skeleton	16
2.4 About the DOM	17
2.5 Preliminary introduction to CSS and JavaScript	19
3 JavaScript	21
3.1 Introduction	21
3.2 Setup	23
3.3 Programming with JS: basis	25
3.4 jQuery	31
3.5 Event listeners	31

4 Shiny, under the hood	33
4.1 Shiny dependencies	33
4.2 Shiny hiddens gems	34
 htmltools	 43
5 htmltools overview	45
5.1 HTML Tags	45
5.2 Notations	45
5.3 Adding new tags	45
5.4 Alternative way to write tags	46
5.5 Playing with tags	46
 6 Dependency utilities	 53
6.1 The dirty approach	53
6.2 The clean approach	55
6.3 Another example: Importing HTML dependencies from other packages	56
6.4 Suppress dependencies	59
 7 Other tools	 61
7.1 CSS	61
 Practice	 65
 8 Template selection	 67
 9 Define dependencies	 69
 10 Template skeleton	 71
 11 Develop custom input widgets	 73
11.1 How does Shiny handle inputs?	73
11.2 How to add new input to Shiny?	73

<i>CONTENTS</i>	5
12 Testing templates elements	75

Prerequisites

- Be familiar with Shiny
- Basic knowledge in HTML and JavaScript is a plus but not mandatory

Disclaimer

This book is not an HTML/Javascript/CSS course! Instead, it provides a *survival kit* to be able to customize Shiny. I am sure however that readers will want to explore more about these topics.

Is this book for me?

You should read this book if you answer yes to the following questions:

- Do you want to know how to develop outstanding shiny apps?
- Have you ever wondered how to develop new input widgets?

Related content

See the RStudio Cloud dedicated project.

```
library(shiny)
library(shinydashboard)
library(shiny.semantic)
library(cascadess)
library(htmltools)
library(purrr)
library(magrittr)
```


Chapter 1

Introduction

There are various Shiny focused resources introducing basic as well as advanced topics such as modules and Javascript/R interactions. However, handling advanced user interfaces was never an emphasis. Clients often desire custom designs, yet this generally exceeds core features of Shiny. We recognized that R App developers lacking a significant background in web development may have found this requirement to be overwhelming. Consequently, the aim of this book is to provide readers the necessary knowledge to extend Shiny's layout, input widgets and output elements. This book is organized into four parts. We first go through the basics of HTML, JavaScript and jQuery. In part 2, we dive into the `{htmltools}` package, providing functions to create and manipulate shiny tags as well as manage dependencies. Part 3 homes in on the development of a new template on top of Shiny by demonstrating examples from the `{bs4Dash}` and `{shinyMobile}` packages, part of the RinterRface project.

Survival Kit

This part will give you basis in HTML, JavaScript to get started...

Chapter 2

HTML

In the following, we will give a short introduction to the HTML language. First of all, let's do the following experience:

- Open your RStudio IDE
- Load shiny with `library(shiny)`
- Execute `p("Hello World")` and notice the output format
- This is an HTML tag!

2.1 HTML Basics

HTML (Hypertext Markup Language) is derived from the SGML (Standard Generalized markup Language). An HTML file contains tags that can be divided into 2 categories:

- paired-tags
- closing-tags

```
<!-- /* paired-tags */ -->
<p></p>
<div></div>

<!-- /* self-closing tags */ -->
<iframe/>
<img/>
<input/>
<br/>
```

2.2 Tag attributes

All tags above don't have any attributes. Yet HTML allows to set attributes inside each tag. There exist a large range of attributes and we will only see 2 of them for now:

- class: may be shared between multiple tags
- id: each must be unique

```
<div class="awesome-item" id="myitem"></div>
<!-- /* the class awesome-item may be applied to multiple tags */ -->
<span class="awesome-item"></span>
```

Both attributes are widely used by CSS and JavaScript (we will discover in the following chapter the jQuery selectors) to apply custom style to a web page. While class may apply to multiple elements, id is restricted to only one item.

2.3 HTML page: skeleton

An HTML page is a collection of tags which will be interpreted by the web browser step by step. The simplest HTML page may be defined as follows:

```
<!DOCTYPE HTML>
<html>
  <head>
    <!-- /* head content here */ -->
  </head>
  <body>
    <!-- /* body content here */ -->
  </body>
</html>
```

- <html> is the may wrapper
- <head> and <body> are the 2 main children

<head> contains dependencies like styles and JavaScript files (but not only), <body> contains the page content. We will see later that JavaScript files are often added just before the end of the <body>.

Only the body content is displayed on the screen!

Let's write the famous Hello World in html:


```
<!DOCTYPE HTML>
<html>
  <head>
    <!-- head content here -->
  </head>
  <body>
    <p>Hello World</p>
  </body>
</html>
```

In order to preview this page in a web browser, you need to save the above snippet to a script `hello-world.html` and double-click on it. It will open with you default web browser.

2.4 About the DOM

The DOM stands for “Document Object Model”, is a convenient representation of the html document. There actually exists multiple DOM types, namely DOM-XML and DOM-HTML but we will only focus on the later (in the following DOM is DOM-HTML). If we consider the last example (Hello World), the associated DOM tree may be inspected in Figure 2.1.

2.4.1 Visualizing the DOM: the HTML inspector

Below, we introduce a tool that we is going to be a valuable ally during our ambitious quest to beautiful shiny user interfaces. In this chapter, we restrict the description to the first panel of the HTML inspector ¹. This feature is available in all web browser but we will only focus on Chrome.

- Open the `hello-world.html` example in a web browser (google chrome here)
- Right-click to open the HTML inspector (developer tools must be enabled if it is not the case)

The HTML inspector is a convenient tool to explore the structure of the current HTML page. On the left-hand side, the DOM tree is displayed and we clearly see that `<html>` is the parent of `<head>` and `<body>`. `<body>` has also 1 child, that is `<p>`. We didn't mention this yet but we can preview any style (CSS) associated to the selected element on the right panel as well as Event Listeners (JavaScript). We will discuss that in the next chapter.

¹As shown in Figure 2.1, the inspector also has tools to debug JavaScript code, inspect files, run performances audit, ... We will describe some of these later in the book.

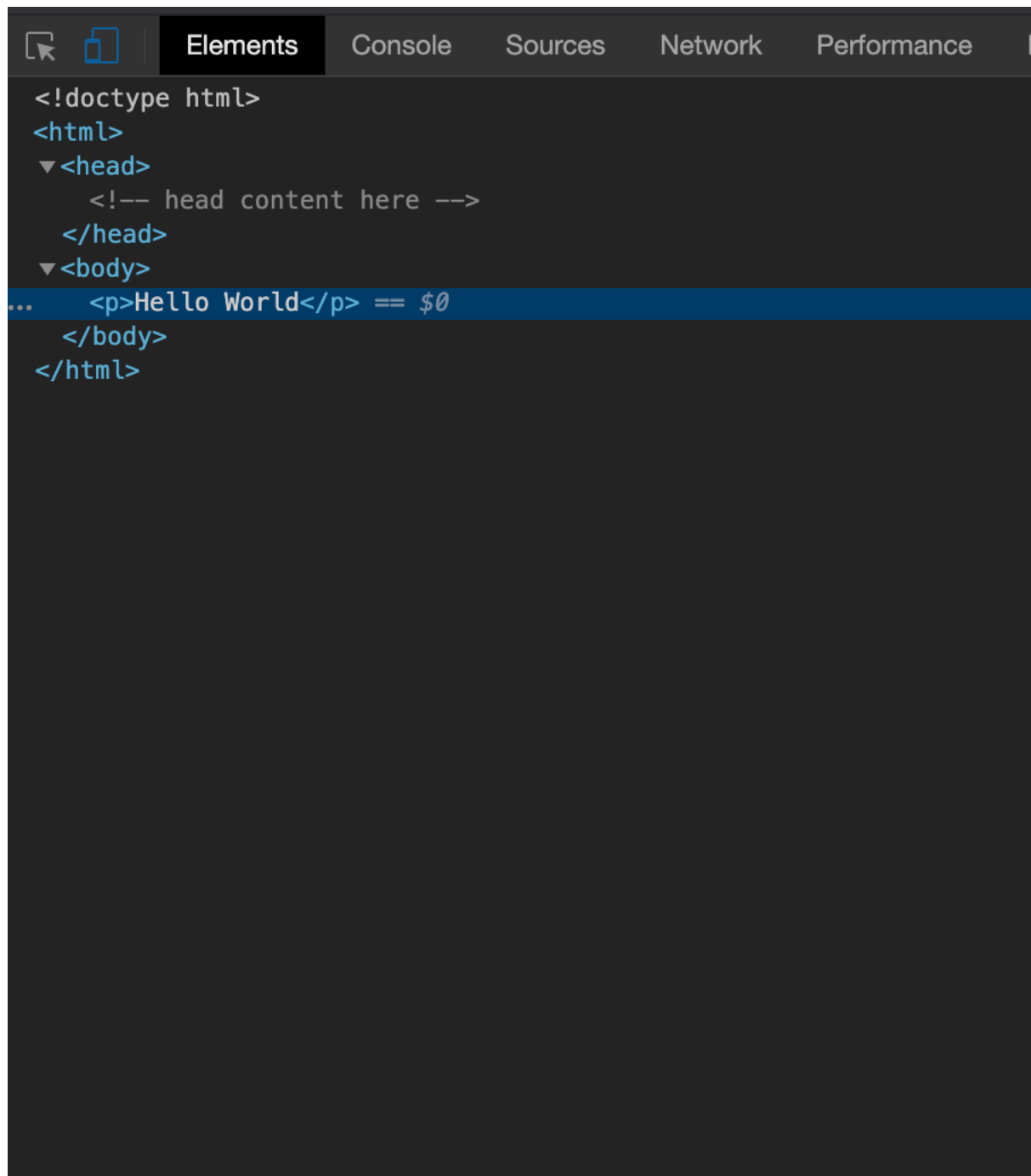


Figure 2.1: Inspection of the DOM in the Hello World example

2.5 Preliminary introduction to CSS and JavaScript

CSS and JavaScript are tools to enhance an HTML page.

2.5.1 HTML and CSS

CSS (Cascading Style Sheets) changes the style of HTML tags by targeting specific classes or ids. For instance, if we want all p tags to have red color we will use:

```
p {  
  color: red;  
}
```

To include CSS in an HTML page, we use the `<style>` tag as follows:

```
<!DOCTYPE HTML>  
<html>  
  <head>  
    <style type="text/css">  
      p {  
        color: red;  
      }  
    </style>  
  </head>  
  <body>  
    <p>Hello World</p>  
  </body>  
</html>
```

You may update the hello-world.html script and run it in your web-browser to see the difference (this is not super impressive but a good start). There exist other ways to include CSS (see next chapters).

2.5.2 HTML and JavaScript

JavaScript is also going to be one of our best friend in this book. You will see how quickly/seamlessly you may add awesome feature to your shiny app.

Let's consider an example below:

```
<!DOCTYPE HTML>
<html>
  <head>
    <style type="text/css">
      p {
        color: red;
      }
    </style>
    <script language="javascript">
      // displays an alert
      alert('Click on the Hello World text!');
      // change text color
      function changeColor(color){
        document.getElementById('hello').style.color = "green";
      }
    </script>
  </head>
  <body>
    <!-- onclick attributes applies the JavaScript function changeColor define above -->
    <p id="hello" onclick="changeColor('green')">Hello World</p>
  </body>
</html>
```

In few lines of code, you can change the color of the text. Wonderful isn't it? Let's move to the next chapter to discover JavaScript!

Chapter 3

JavaScript

3.1 Introduction

JavaScript (JS) was created in 1995 by Brendan Eich and also known as ECMAScript (ES). Interestingly, you might have heard about ActionScript, which is no more than an implementation of ES by Adobe Systems. Nowadays, JavaScript is a centerpiece of the web and included in almost all websites.

Let's make a little experiment. If you have a personal blog (it is very popular in the RStats community) you probably know Hugo or Jekyll. These tools allow to quickly setup professional looking (or at least not too ugly) blogs in literally few minutes. You can focus on the content and this is what matters! Now, if you open the HTML inspector introduced in Chapter 2, click on the elements tab (in theory it is the first tab and open by default), and uncollapse the `<head>` tag, you see that a lot of scripts are included, as shown in Figure 3.1. Same remark in the `<body>` tag.

There are 2 ways to include scripts:

- Use the `<script>` tag with the JS code inside
- Import an external file containing the JS code and only

```
<script type="text/javascript">  
// JS code here  
</script>
```

```
<!-- We use the src attribute to link the external file -->  
<script type="text/javascript" src="file.js">
```

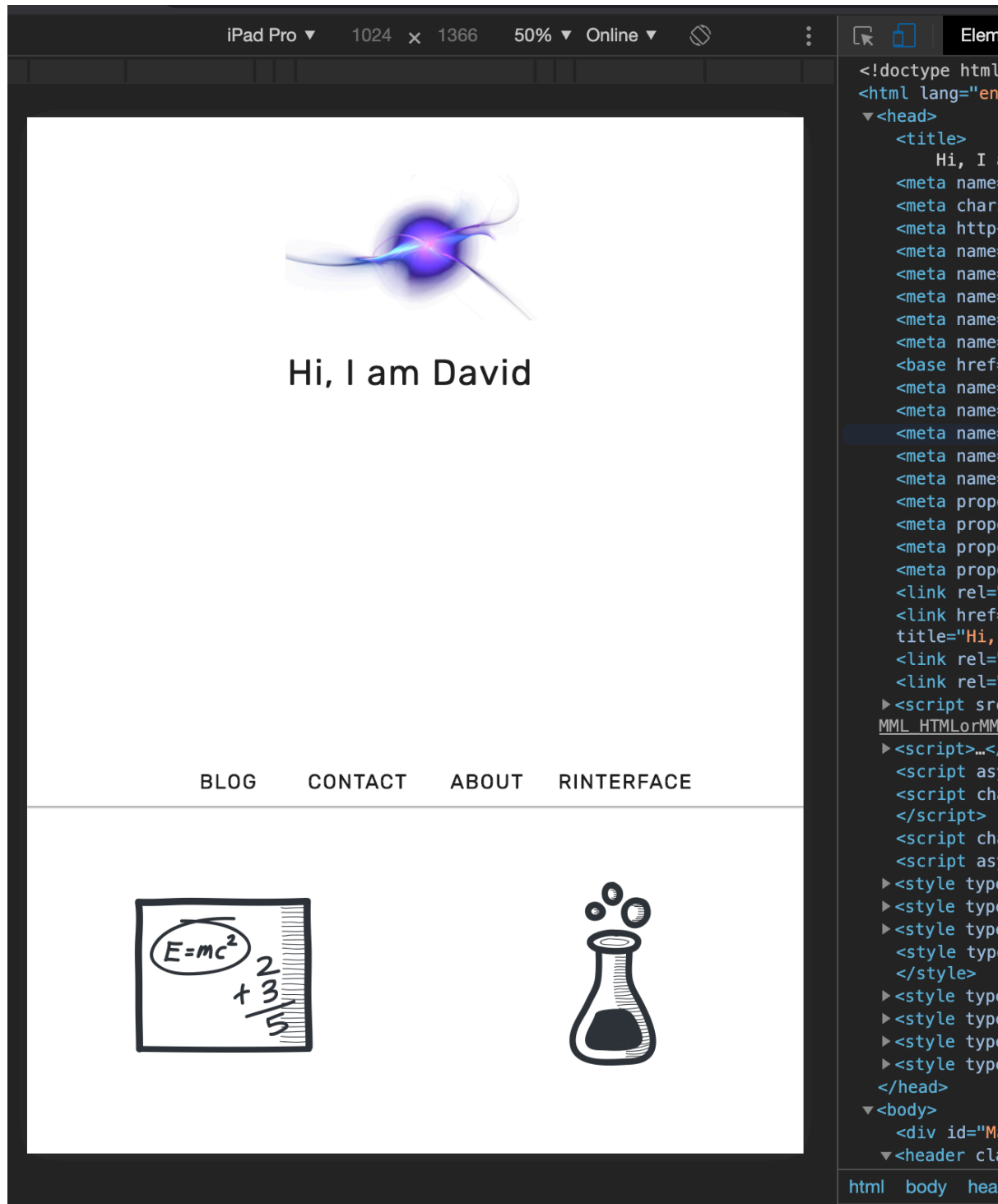


Figure 3.1: A website is full of JavaScript

Whether to choose the first or second method depends on the content of your script. If we consider jQuery, a well known JS library, it contains so much lines of code that it does not make sense to select the first method.

3.2 Setup

Like R or Python, JavaScript is an interpreted language. It is also executed client side, that is in the navigator. It also means that you cannot run js code without suitable tools.

3.2.1 Node

Node contains an interpreter for JS as well as a dependencies manager, npm (Node Package Manager). To install Node on your computer, browse to the website and follow the instruction. Once done, open a terminal and check if

```
$ which node
$ node --version
```

returns something. If not, it means that Node is not properly installed.

3.2.2 Choose a good IDE

I really like VSCode for all the JS things since it contains a Node interpreter and you can seamlessly execute any JS code (the truth is because I'm a big fan of the dracula color theme). But the Rstudio IDE may also be fine, provided that you have Node installed. Below, we will see how to run a JS code in both IDE.

3.2.3 First Script

Let's write our first script:

```
console.log("Hello World");
```

You notice that all instruction end by ;. You can run this script either in Rstudio IDE or VSCode.

In VSCode, clicking on the run arrow (top center) of Figure 3.2, triggers the `node hello.js` command, which tells Node to run my script. We see the result in the right panel (code=0 means the execution is fine and we even have the

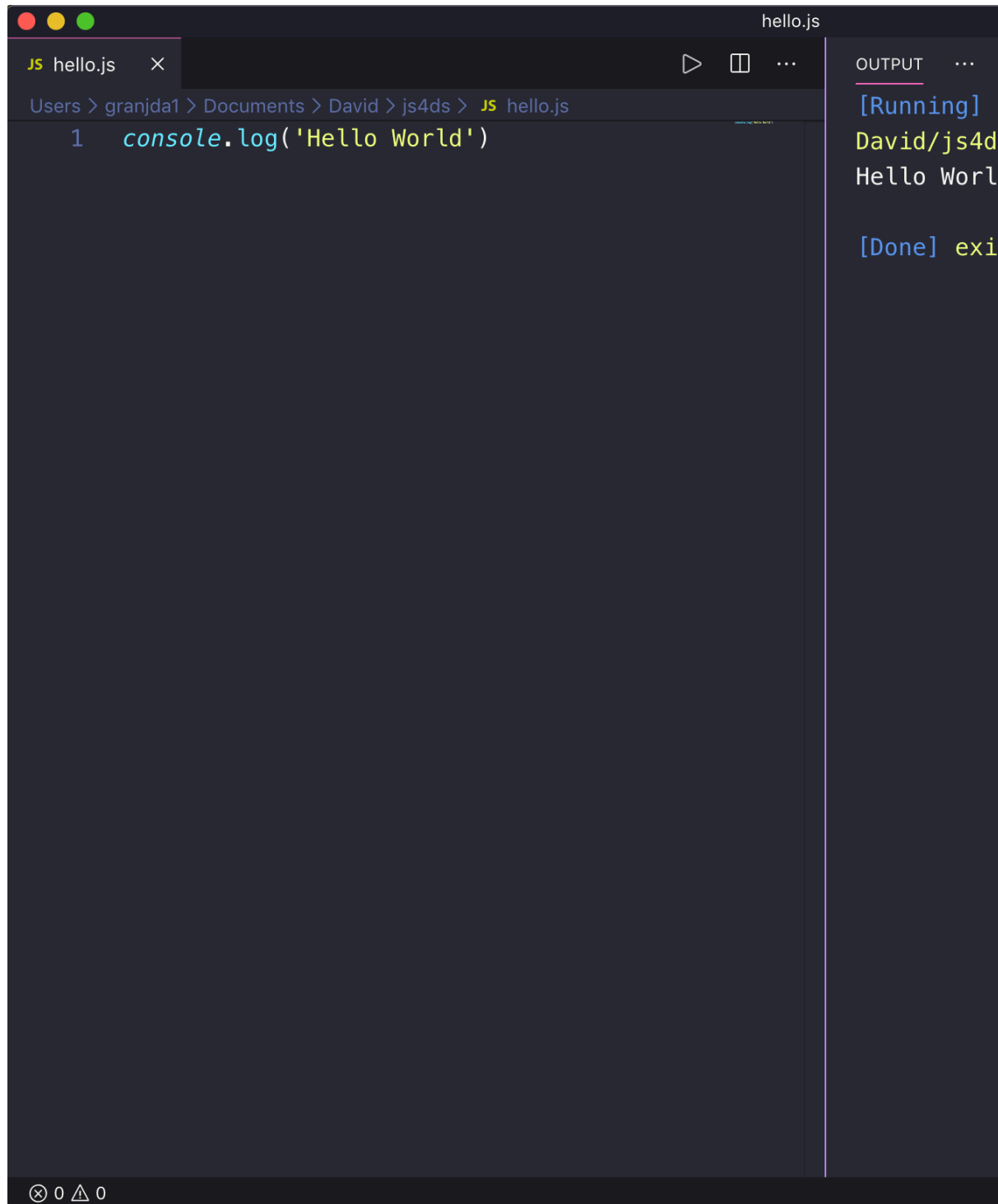
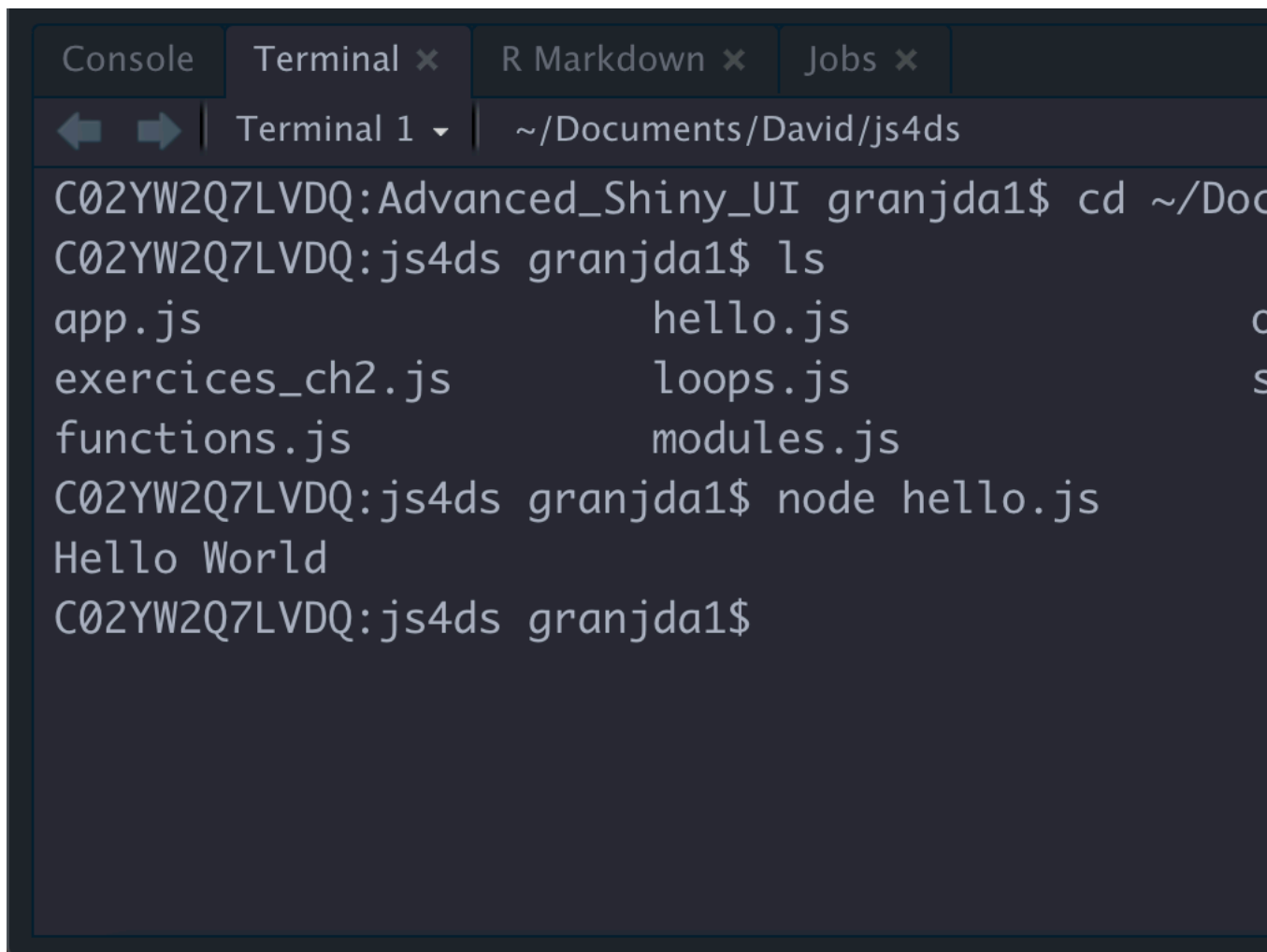


Figure 3.2: Run JS in VSCode

compute time). To run this script in the RStudio IDE, you need to click on the terminal tab (you could also open a basic terminal) and type `node hello.js` (or `node mycustompath/hello.js` if you are not in the folder containing the script). You should see the Hello World message in the console (see Figure 3.3).

A screenshot of the RStudio IDE's terminal window. The terminal has tabs for 'Console', 'Terminal', 'R Markdown', and 'Jobs'. The 'Terminal' tab is active, showing 'Terminal 1' with the path '~/Documents/David/js4ds'. The terminal output shows the following commands and results:

```
C02YW2Q7LVDQ:Advanced_Shiny_UI granjda1$ cd ~/Doc
C02YW2Q7LVDQ:js4ds granjda1$ ls
app.js                hello.js
exercices_ch2.js      loops.js
functions.js          modules.js
C02YW2Q7LVDQ:js4ds granjda1$ node hello.js
Hello World
C02YW2Q7LVDQ:js4ds granjda1$
```

Figure 3.3: Run JS in a terminal

3.3 Programming with JS: basis

We are now all set to introduce the basis of JS. As many languages, JS is made of variables and instructions (We saw above that instructions end by `;`).

3.3.1 JS types

JS defines several types:

- Number: does not distinguish between integers and others (in R for instance, numeric contains integers and double)
- String: characters ('blabla')
- Boolean: true/false

To check the type of an element, we may use the `typeof` operator (this is not a function like the `typeof` function in R).

```
typeof 1; // number
typeof 'pouic'; // string
```

3.3.2 Variables

A variable is defined by:

- a type
- a name
- a value

Valid variable names:

don't use an existing name like `typeof`

don't start with a number (123)

don't include any space (total price)

Based on the above forbidden items, you can use the camelCase syntax to write your variables in JS. To set a variable we use `let` (there exists `var` but this is not the latest JS norm (ES6). You will see later that we still use `var` in the shiny core and many other R packages):

```
let myVariable = 'welcome';
console.log(myVariable);
```

Then we can use all mathematical operators to manipulate a variable.

```
let myNumber = 1; // affectation
myNumber--; // decrement
console.log(myNumber); // print 0
```

List of numerical operators in JS:

+

-

*

/

% (modulo)

++ (incrementation)

-- (decrementation)

To concatenate 2 strings, use +.

3.3.3 Conditions

Below are the operators to check conditions.

== (A equal B)

!= (A not equal to B)

(>=)

< (<=)

AND (A AND B)

OR (A OR B)

To test conditions there exists several ways:

- `if (condition) { console.log('Test passed'); }`
- `if (condition) { instruction A } else { instruction B }`

This is very common to other languages (and R for instance). Whenever a lot of possible conditions need to be evaluated, it is better to choose the **switch**.

```
switch (variable) {
  case val1: // instruction 1
    break; // don't forget the break!
  case val2: // instruction 2
    break;
  default: // when none of val1 and val2 are satisfied
}
```

3.3.4 Iterations

Iterations allow to repeat an instruction or a set of instructions multiple times.

3.3.4.1 For loops

The for loop has multiple ways to be used. Below is the most classic. We start by defining the index (variable). We then set an upper bound and we finish by incrementing the index value. We execute the instruction between curly braces.

```
const max = 10; // we never mentionned constants before. This is the way to call them
for (let i = 0; i <= max; i++) {
  console.log(i); // this will print i 10 times
}
```

Contrary to R, JavaScript index starts from 0 (not from 1)! This is good to keep in mind when we will mix both R and JS.

Below is another way to create a for loop:

```
let samples = ['blabla', 1, null]; // this is an array!
for (let sample of samples) {
  console.log(sample);
}
```

What is the best for loop? The answer is: it depends on the situation! Actually, there even exists other ways (replace of by in and you get the indexes of the array, like the with the first code, but this is really not recommended).

3.3.4.2 Other iterations: while, do while

We will clearly never use them in this book but this is good to know.

```
const h = 3; i = 0;
while (i <= h) {
  console.log(i);
  i++; // we need to increment to avoid infinite loop
}
```

3.3.5 Objects

JavaScript is an object oriented programming language (like Python). An object is defined by:

- a type
- some properties
- some methods (to manipulate properties)

Let's define our first object below:

```
const me = {
  name : 'Divad',
  age : 29,
  music : '',
  printName: function() {
    console.log(`I am ${this.name}`);
  }
}

console.log(JSON.stringify(me)); // print a human readable object.

console.log(me.name);
console.log(me.age);
console.log(me.music);
// don't repeat yourself!!!
for (let key in me) { // here is it ok to use `in`
  console.log(`me[${key}] is ${me[key]}`);
}

me.printName();
```

Some comments on the above code:

- to access an object property, we use `object.property`
- to print a human readable version of the object `JSON.stringify` will do the job
- we introduced string interpolation with `${*}`. `*` may be any valid expression.
- methods are accessed like properties (we may also pass parameters). We use `this` to refer to the object itself. Take note, we will see it a lot!

In JavaScript, we can find already predefined objects to interact with arrays, dates.

3.3.6 Functions

Functions are useful to wrap a succession of instructions to accomplish a given task. Defining functions allows programmers to save time (less copy and paste,

less search and replace), do less errors and share code more easily. In modern JavaScript (ES6), functions are defined as follows:

```
const a = 1;
const fun = (parm1, parm2) => {
  console.log(a);
  let p = 3;
  return Math.max(parm1, parm2); // I use the Math object that contains the max method
}
let res = fun(1, 2);
console.log(res); // prints a and 2. a global
console.log(p); // fails because p was defined inside the function
```

This functions computes the maximum of 2 provided numbers. Some comments about scoping rules: variables defined inside the function are available for the function but not outside. The function may use global variables defined outside of it.

3.3.6.1 Export functions: about modules

What happens if you wrote 100 functions and you would like to reuse some of them in different scripts? To prevent copy and pasting, we will introduce modules. Let's save the below function in a script `utils.js`:

```
const findMax = (parm1, parm2) => {
  return Math.max(parm1, parm2); // I use the Math object that contains the max method
}

module.exports = {
  findMax = findMax
}
```

Now, if we create a `test.js` script in the same folder and want to use `findMax`, we need to import the corresponding module:

```
const {findMax} = require('./utils.js');
findMax(1, 2); // prints 2
```

3.4 jQuery

3.5 Event listeners

In the next chapters, we will see that some of the underlying JS code to build custom shiny inputs share the same utils functions. Therefore, introducing modules is necessary.

Chapter 4

Shiny, under the hood

In the 2 previous chapters, we quickly introduced HTML and JavaScript. Chapter 1 finished on an example showing how to modify an HTML page with JavaScript. Yet, in Chapter 2, we simply introduced the language without showing any link with HTML. In this chapter we are going to see what shiny has under the hood to better understand the link between those languages. We will particularly accord a great importance to jQuery.

4.1 Shiny dependencies

This book assumes you are already quite advanced in Shiny. Below we will investigate elements that you probably never noticed.

Shiny allows to develop web applications by only using R. If you remember about the first experiment of Chapter 2, we only did

```
library(shiny)
p("Hello World")
```

Hello World

to notice that the `p` function generates HTML. We will see in the next chapters other tools to build/modify/delete tags. The main difference between HTML tags and Shiny tags is the absence of closing tag for Shiny. For instance, in raw HTML, we expect `<p>` to be closed by `</p>`. In Shiny, we only call `p(...)`, where `...` may be attributes like `class/id` or children tags. However, this is still a bit far from web application since there is no user interface, interactivity and computations. The simplest Shiny layout is the `fluidPage` (if you type `shinyapp` in the R console, it will show a predefined snippet with this default template):

```
ui <- fluidPage(  
  p("Hello World")  
)  
  
server <- function(input, output, session) {}  
shinyApp(ui, server)
```

At first glance, the page only contains the Hello World text. Waiiit ... are you sure about this? Let's run the above example and open the HTML inspector. Results are displayed on Figure 4.1. In Chapter 6 we will see better tools to extract HTML dependencies.

We see in the head section that Shiny has 4 dependencies:

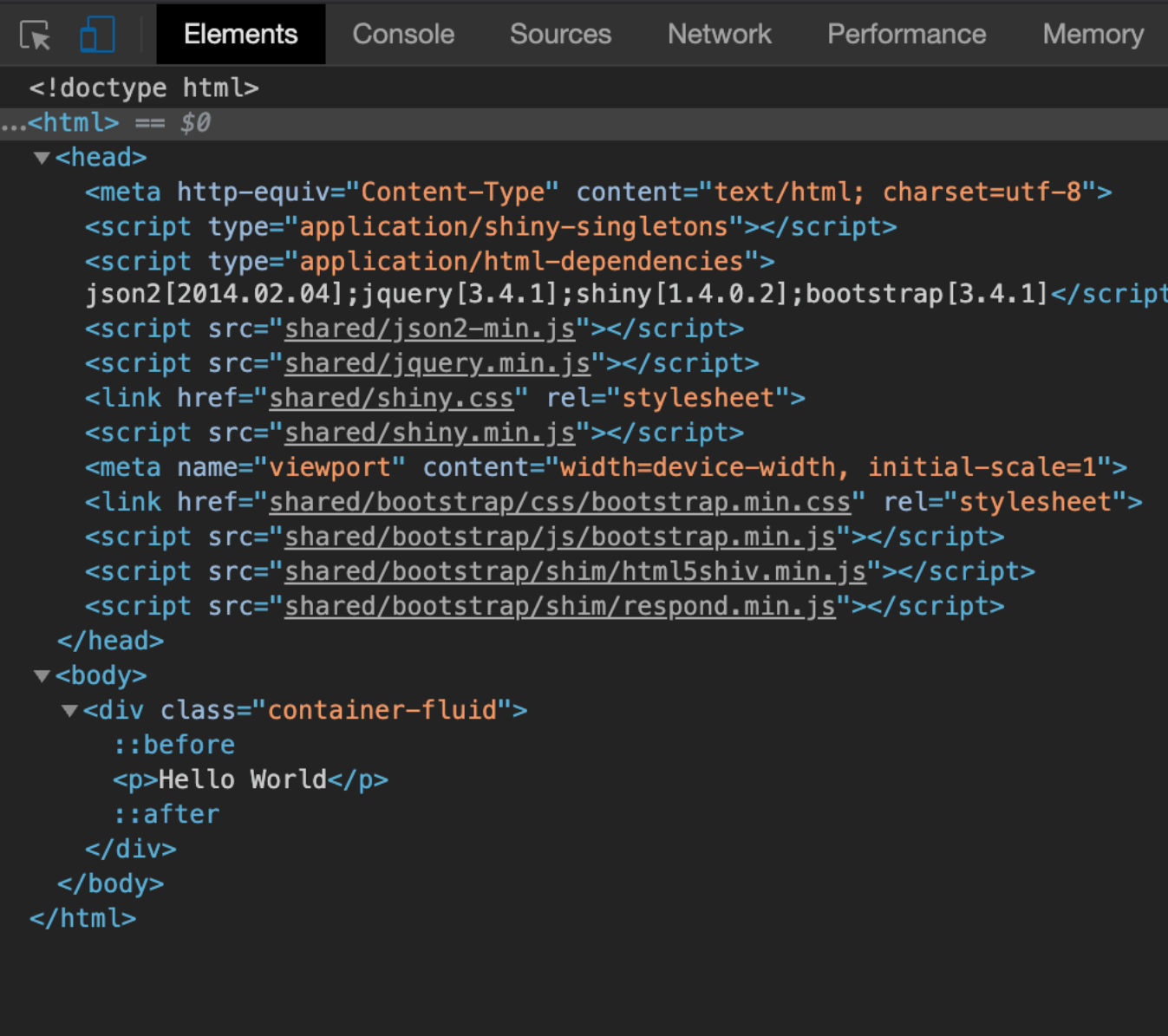
- json2
- jQuery 3.4.1
- shiny (custom JavaScript and CSS)
- Bootstrap 3.4.1 (JavaScript and CSS) + other files (html5shiv, respond)

Bootstrap is here to provide plug and play design and interactions (tabs, navs). For instance the `fluidRow` and `column` functions of Shiny leverage the Bootstrap grid to control how elements are displayed in a page. This is convenient because it avoids to write a crazy amount of CSS/JavaScript and always reinvent the wheel. jQuery is a famous JavaScript library providing a user friendly interface to manipulate the DOM and is present in almost all actual websites. It is slightly easier (understand more convenient to use) than vanilla JS, even though web developers tend to avoid it to go back to vanilla JS (Bootstrap 5, the next iteration of Bootstrap will not rely on jQuery anymore). Shiny has its own JS and CSS files (we will discuss this very soon). Finally, json2 is a library to handle the JSON data format (JavaScript Object Notation). In the following chapters we will use it a lot, through the jsonlite package that allows to transform JSON objects in R objects and inversely.

In summary, all those libraries are necessary to make Shiny what it is! Customizing Shiny will imply to alter those existing libraries (except the Shiny core JavaScript and json2).

4.2 Shiny hiddens gems

As promised earlier, let's talk about the Shiny JavaScript core. The goal of this part is to better understand the mechanisms behind Shiny, especially the input system.



```
<!doctype html>
...<html> == $0
  <head>
    <meta http-equiv="Content-Type" content="text/html; charset=utf-8">
    <script type="application/shiny-singletons"></script>
    <script type="application/html-dependencies">
      json2[2014.02.04];jquery[3.4.1];shiny[1.4.0.2];bootstrap[3.4.1]</script>
    <script src="shared/json2-min.js"></script>
    <script src="shared/jquery.min.js"></script>
    <link href="shared/shiny.css" rel="stylesheet">
    <script src="shared/shiny.min.js"></script>
    <meta name="viewport" content="width=device-width, initial-scale=1">
    <link href="shared/bootstrap/css/bootstrap.min.css" rel="stylesheet">
    <script src="shared/bootstrap/js/bootstrap.min.js"></script>
    <script src="shared/bootstrap/shim/html5shiv.min.js"></script>
    <script src="shared/bootstrap/shim/respond.min.js"></script>
  </head>
  <body>
    <div class="container-fluid">
      ::before
      <p>Hello World</p>
      ::after
    </div>
  </body>
</html>
```

Figure 4.1: Shiny dependencies

4.2.1 Initialization

Upon initialization, Shiny runs several JavaScript functions. Not surprisingly, there is one called `initShiny` containing a substantial amount of elements.

We find utils functions like `bindOutputs`, `unbindOutputs` to respectively bind/unbind outputs, `bindInputs` and `unbindInputs` for inputs. Only `bindAll` and `unbindAll` are available to the user (see a usecase here). To illustrate what they do, let's run the app below.

```
ui <- fluidPage(  
  sliderInput("obs", "Number of observations:",  
    min = 0, max = 1000, value = 500  
  ),  
  plotOutput("distPlot")  
)  
  
server <- function(input, output, session) {  
  output$distPlot <- renderPlot({  
    hist(rnorm(input$obs))  
  })  
}  
shinyApp(ui, server)
```

We then open the HTML inspector and run `Shiny.unbindAll(document)` (document is the scope, that is where to search). Try to change the slider input. What do you observe? Now let's type `Shiny.bindAll(document)` and update the slider value. What happens? Magic isn't it? This simply shows that when inputs are not bound, nothing happens so binding inputs is necessary. Let's see below how an input binding works.

4.2.1.1 Input bindings

The input binding is defined by a class living in the `input_binding.js` file. An input binding allows Shiny to identify each instance of a given input and what you can do with this input. The interesting thing is that if your app contains 10 different sliders, they all share the same input binding! An input binding is an object having the following methods:

- `find(scope)`: this method specifies how to find the current input element (el) in the DOM. scope refers to the `document` element. In general, we use jQuery selector to search for a class.
- `initialize(el)`: This is called before the input is bound but not all input need to be initialized. Some API like Framework7 require to almost always have an initialize method (We will see later).

- `getValue(el)`: returns the input value. The way to obtain the value tightly depends on the object and is different for almost all inputs.
- `setValue(el, value)`: This method is used to set the value of the current input.
- `receiveMessage(el, data)`: This method is the JavaScript part of all the `updateInput` functions. We usually call the `setValue` method inside.
- `subscribe(el, callback)`: We listen to events telling under which circumstances to change the input value. Some API like Bootstrap explicitly mention those events (like `hide.bs.tab`, `shown.bs.tab`, ...).
- `getRatePolicy`: when `callback` is true in the `subscribe` method, we apply a specific rate policy (debounce, throttle). This is useful for instance when we don't want to flood the server with useless update requests. For a slider, we only want to send the value as soon as the range stops moving and not all intermediate values. Those elements are defined here.

At the end of the input binding definition, we register it for Shiny.

```
let myBinding = new Shiny.inputBinding();
$.extend(myBinding, {
  // methods go here
});

Shiny.inputBindings.register(myBinding, 'reference');
```

Even though the Shiny documentation mentions a `Shiny.inputBindings.setPriority` method to handle conflicting binding, it is better not to use it.

Upon initialization, Shiny calls the `initializeInputs` function that takes all input bindings and call their `initialize` method before binding all inputs. Note that once an input has been initialized it has a `_shiny_initialized` tag to avoid initializing it twice. As shown above, the `initialize` method is not always defined.

4.2.2 Utilities to quickly define new inputs

If you ever wondered where the `Shiny.onInputChange` or `Shiny.setInputValue` comes from (see article), it is actually defined in the `initShiny` function.

```
exports.setInputValue = exports.onInputChange = function(name, value, opts) {
  opts = addDefaultInputOpts(opts);
  inputs.setInput(name, value, opts);
};
```

Briefly, this function avoids to create an input binding. It is faster to code but there is a price to pay: you lose the possibility to easily update the new input.

Indeed, all input functions like `sliderInput` have their own update function like `updateSliderInput`, because of the custom input binding system (We will see it very soon)!

4.2.3 Get access to initial values

Something we may notice when exploring the `initShiny` function is the existence of a shinyapp object, defined as follows:

```
var shinyapp = exports.shinyapp = new ShinyApp();
```

Let's explore what `ShinyApp` contains. The definition is located in the `shinyapps.js` script.

```
var ShinyApp = function() {  
  this.$socket = null;  
  
  // Cached input values  
  this.$inputValues = {};  
  
  // Input values at initialization (and reconnect)  
  this.$initialInput = {};  
  
  // Output bindings  
  this.$bindings = {};  
  
  // Cached values/errors  
  this.$values = {};  
  this.$errors = {};  
  
  // Conditional bindings (show/hide element based on expression)  
  this.$conditionals = {};  
  
  this.$pendingMessages = [];  
  this.$activeRequests = {};  
  this.$nextRequestId = 0;  
  
  this.$allowReconnect = false;  
};
```

It creates several properties, some of them are easy to guess like `inputValues` or `initialInput`. Let's run the example below and open the HTML inspector. Notice that the `sliderInput` is set to 500 at `t0` (initialization).

```
ui <- fluidPage(  
  sliderInput("obs", "Number of observations:",  
    min = 0, max = 1000, value = 500  
  ),  
  plotOutput("distPlot")  
)  
  
server <- function(input, output, session) {  
  output$distPlot <- renderPlot({  
    hist(rnorm(input$obs))  
  })  
}  
shinyApp(ui, server)
```

Figure 4.2 shows how to access Shiny's initial input value with `Shiny.shinyapp.$initialInput.obs`. After changing the slider position, its value is given by `Shiny.shinyapp.$inputValues.obs`. `$initialInput` and `$inputValues` contains way more elements but we are only interested by the slider in this example.

I acknowledge, the practical interest might be limited but still good to know for debugging purposes.

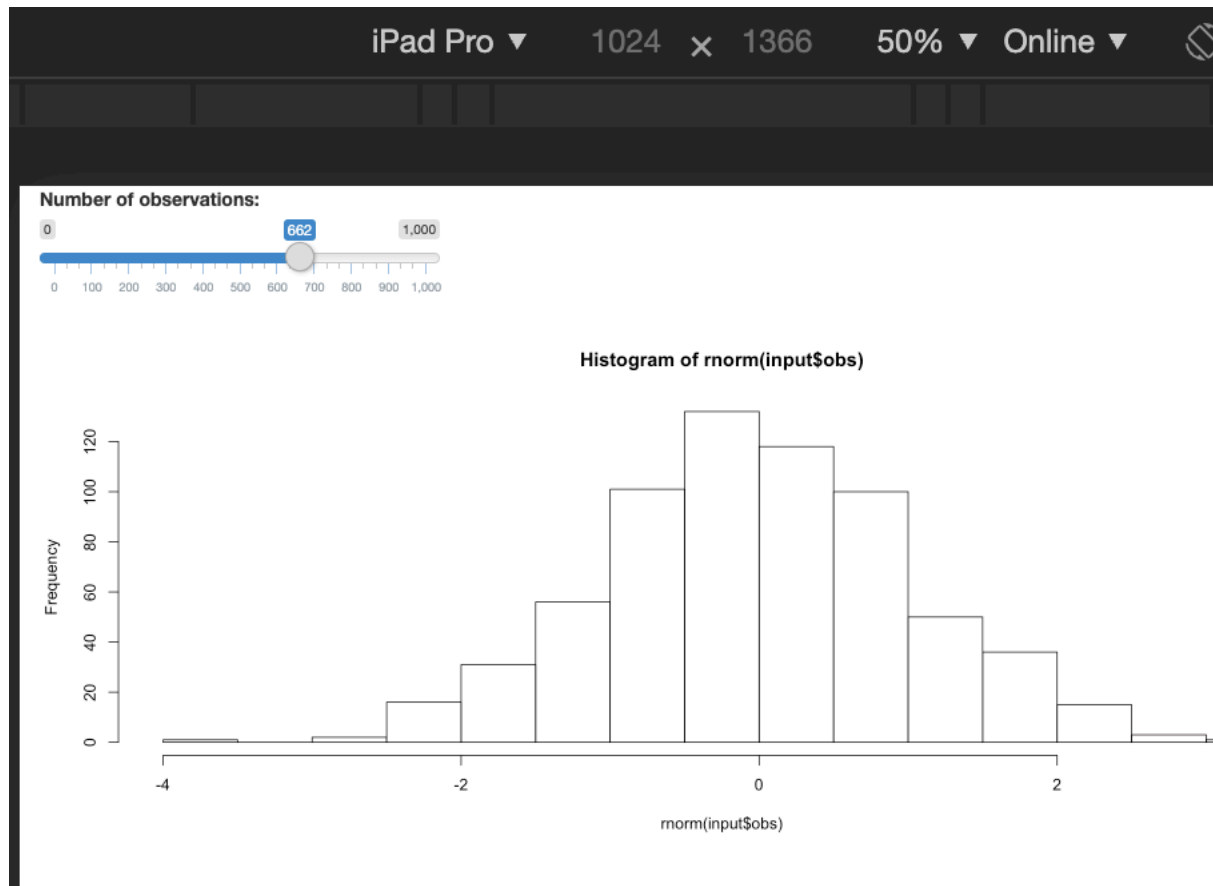


Figure 4.2: Explore initial input values

htmltools

While building a custom html template, you will need to know more about the wonderful `htmltools` developed by Winston Chang, member of the shiny core team. It has the same spirit as `devtools`, that is, making your web developer life easier. What follows does not have the pretention to be an exhaustive guide about this package. Yet, it will provide you with the main tools to be more efficient.

Chapter 5

htmltools overview

5.1 HTML Tags

htmltools contains tools to write HTML tags we saw in Chapter 2:

```
div()
```

If you had to gather multiple tags together, prefer `tagList()` as `list()`, although the HTML output is the same. The first has the `shiny.tag.list` class in addition to `list`. (The `Golem` package allows to test if a R object is a tag list, therefore using `list` would make the test fail).

5.2 Notations

Whether to use `tags$div` or `div` depends if the tag is exported by default. For instance, you could use `htmltools::div` but not `htmltools::nav` since `nav` does not have a dedicated function (only for `p`, `h1`, `h2`, `h3`, `h4`, `h5`, `h6`, `a`, `br`, `div`, `span`, `pre`, `code`, `img`, `strong`, `em`, `hr`). Rather use `htmltools::tags$nav`. Alternatively, there exists a function (in `shiny` and `htmltools`) called `withTags()`. Wrapping your code in this function enables you to use `withTags(nav(), ...)` instead of `tags$nav()`.

5.3 Adding new tags

The `tag` function allows to add extra HTML tags not already defined. You may use it as follows:

```

tag("test", list(class = "test", p("Custom Tag")))
# structure below
tag
  "test"
  list
    class = "test"
    p
      "Custom Tag"

```

5.4 Alternative way to write tags

htmltools comes with the `HTML()` function that you can feed with raw HTML:

```

HTML('<div>Blabla</div>')
# will render exactly like
div("Blabla")

# but there class is different
class(HTML('<div>Blabla</div>'))
class(div("Blabla"))

```

You will not be able to use tag related functions, as in the following parts. Therefore, I strongly recommend using R and not mixing HTML in R. Interestingly, if you want to convert HTML to R code, there is a Shiny App developed by Alan Dipert from RStudio, namely `html2R`. There are some issues, non standard attributes (like `data-toggle`) are not correctly processed but there are fixes. This will save you precious time!

5.5 Playing with tags

5.5.1 Tags structure

According to the `tag` function, a tag has:

- a name such as `span`, `div`, `h1` ... `tag$name`
- some attributes, which you can access with `tag$attribs`
- children, which you can access with `tag$children`
- a class, namely `"shiny.tag"`

For instance:

```
# create the tag
myTag <- div(
  class = "divclass",
  id = "first",
  h1("Here comes your baby"),
  span(class = "child", id = "baby", "Crying")
)
# access its name
myTag$name
# access its attributes (id and class)
myTag$attrs
# access children (returns a list of 2 elements)
myTag$children
# access its class
class(myTag)
```

How to modify the class of the second child, namely span?

```
second_children <- myTag$children[[2]]
second_children$attrs$class <- "adult"
myTag
# Hummm, this is not working ...
```

Why is this not working? By assigning `myTag$children[[2]]` to `second_children`, `second_children$attrs$class <- "adult"` modifies the class of the copy and not the original object. Thus we do:

```
myTag$children[[2]]$attrs$class <- "adult"
myTag
```

In the following section we explore helper functions, such as `tagAppendChild` from `htmltools`.

5.5.2 Useful functions for tags

`htmltools` and `Shiny` have powerful functions to easily add attributes to tags, check for existing attributes, get attributes and add other siblings to a list of tags.

5.5.2.1 Add attributes

- `tagAppendAttributes`: this function allow you to add a new attribute to the current tag.

For instance, assuming you created a div for which you forgot to add an id attribute:

```
mydiv <- div("Where is my brain")
mydiv <- tagAppendAttributes(mydiv, id = "here_it_is")
```

You can pass as many attributes as you want, including non standard attributes such as `data-toggle` (see Bootstrap 3 tabs for instance):

```
mydiv <- tagAppendAttributes(mydiv, `data-toggle` = "tabs")
# even though you could proceed as follows
mydiv$attrs[["aria-controls"]] <- "home"
```

5.5.2.2 Check if tag has specific attribute

- `tagHasAttribute`: to check if a tag has a specific attribute

```
# I want to know if div has a class
mydiv <- div(class = "myclass")
has_class <- tagHasAttribute(mydiv, "class")
has_class
# if you are familiar with %>%
has_class <- mydiv %>% tagHasAttribute("class")
has_class
```

5.5.2.3 Get all attributes

- `tagGetAttribute`: to get the value of the targeted attributes, if it exists, otherwise `NULL`.

```
mydiv <- div(class = "test")
# returns the class
tagGetAttribute(mydiv, "class")
# returns NULL
tagGetAttribute(mydiv, "id")
```

5.5.2.4 Set child/children

- `tagSetChildren` allows to create children for a given tag. For instance:


```
mydiv <- div(class = "parent", id = "mother", "Not the mama!!!")
# mydiv has 1 child "Not the mama!!!"
mydiv
children <- lapply(1:3, span)
mydiv <- tagSetChildren(mydiv, children)
# mydiv has 3 children, the first one was removed
mydiv
```

Notice that `tagSetChildren` removes all existing children. Below we see another set of functions to add children while conserving existing ones.

5.5.2.5 Add child or children

- `tagAppendChild` and `tagAppendChildren`: add other tags to an existing tag. Whereas `tagAppendChild` only takes one tag, you can pass a list of tags to `tagAppendChildren`.

```
mydiv <- div(class = "parent", id = "mother", "Not the mama!!!")
otherTag <- span("I am your child")
mydiv <- tagAppendChild(mydiv, otherTag)
```

You might wonder why there is no `tagRemoveChild` or `tagRemoveAttributes`. Let's look at the `tagAppendChild`

```
tagAppendChild <- function (tag, child) {
  tag$children[[length(tag$children) + 1]] <- child
  tag
}
```

Below we write the `tagRemoveChild`, where `tag` is the target and `n` is the position to remove in the list of children:

```
mydiv <- div(class = "parent", id = "mother", "Not the mama!!!", span("Hey!"))

# we create the tagRemoveChild function
tagRemoveChild <- function(tag, n) {
  # check if the list is empty
  if (length(tag$children) == 0) {
    stop(paste(tag$name, "does not have any children!"))
  }
  tag$children[n] <- NULL
  tag
}
```

```
mydiv <- tagRemoveChild(mydiv, 1)
mydiv
```

When defining the `tagRemoveChild`, we choose `[]` instead of `[[` to allow to select multiple list elements:

```
mydiv <- div(class = "parent", id = "mother", "Not the mama!!!", "Hey!")
# fails
`[[`(mydiv$children, c(1, 2))
# works
`[`(mydiv$children, c(1, 2))
```

Alternatively, we could also create a `tagRemoveChildren` function. Also notice that the function raises an error if the provided tag does not have children.

5.5.3 Other interesting functions

The Golem package written by thinkr contains neat functions to edit your tags. Particularly, the `tagRemoveAttributes`:

```
tagRemoveAttributes <- function(tag, ...) {
  attrs <- as.character(list(...))
  for (i in seq_along(attrs)) {
    tag$attribs[[ attrs[i] ]] <- NULL
  }
  tag
}
```

```
mydiv <- div(class = "test", id = "coucou", "Hello")
tagRemoveAttributes(mydiv, "class", "id")
```

5.5.4 Conditionally set attributes

Sometimes, you only want to set attributes under specific conditions.

```
my_button <- function(color = NULL) {
  tags$button(
    style = paste("color:", color),
    p("Hello")
  )
}

my_button()
```

This example will not fail but having `style="color: "` is not clean. We may use conditions:

```
my_button <- function(color = NULL) {
  tags$button(
    style = if (!is.null(color)) paste("color:", color),
    p("Hello")
  )
}

my_button("blue")
my_button()
```

In this example, style won't be available if color is not specified.

5.5.5 Using %>%

While doing a lot of manipulation for a tag, if you don't need to create intermediate objects, this is a good idea to use %>% from magrittr:

```
div(class = "cl", h1("Hello")) %>%
  tagAppendAttributes(id = "myid") %>%
  tagAppendChild(p("some extra text here!"))
```

5.5.6 Programmatically create children elements

Assume you want to create a tag with 3 children inside:

```
div(
  span(1),
  span(2),
  span(3),
  span(4),
  span(5)
)
```

The structure is correct but imagine if you had to create 1000 `span` or fancier tag. The previous approach is not consistent with DRY programming. `lapply` function will be useful here (or the purrr `map` family):

```
# base R
div(lapply(1:5, function(i) span(i)))
# purrr + %>%
map(1:5, function(i) span(i)) %>% div()
```


Chapter 6

Dependency utilities

When creating a new template, you sometimes need to import custom HTML dependencies that do not come along with shiny. No problem, `htmltools` is here for you (shiny also contains these functions).

6.1 The dirty approach

Let's consider the following example. I want to include a bootstrap 4 card in a shiny app. This example is taken from an interesting question [here](#). The naive approach would be to include the HTML code directly in the app code

```
# we create the card function before
my_card <- function(...) {
  withTags(
    div(
      class = "card border-success mb-3",
      div(class = "card-header bg-transparent border-success"),
      div(
        class = "card-body text-success",
        h3(class = "card-title", "title"),
        p(class = "card-text", ...)
      ),
      div(class = "card-footer bg-transparent border-success", "footer")
    )
  )
}

# we build our app
shinyApp(
```

```

ui = fluidPage(
  fluidRow(
    column(
      width = 6,
      align = "center",
      br(),
      my_card("blablabla. PouetPouet Pouet.")
    )
  ),
  server = function(input, output) {}
)

```

and desperately see that nothing is displayed. If you remember, this was expected since shiny does not contain bootstrap 4 dependencies and this card is unfortunately a bootstrap 4 object. Don't panic! We just need to tell shiny to load the css we need to display this card (if required, we could include the javascript as well). We could use either `includeCSS()`, `tags$head(tags$link(rel = "stylesheet", type = "text/css", href = "custom.css"))`. See more here.

```

shinyApp(
  ui = fluidPage(
    # load the css code
    includeCSS(path = "https://maxcdn.bootstrapcdn.com/bootstrap/4.0.0/css/bootstrap.m
  fluidRow(
    column(
      width = 6,
      align = "center",
      br(),
      my_card("blablabla. PouetPouet Pouet.")
    )
  ),
  server = function(input, output) {}
)

```

The card is ugly (which is another problem we will fix later) but at least displayed.

When I say this approach is dirty, it is because it will not be easily re-usable by others. Instead, we prefer a packaging approach, like in the next section.

6.2 The clean approach

We will use the `htmlDependency` and `attachDependencies` functions from `htmltools`. The `htmlDependency` takes several arguments:

- the name of your dependency
- the version (useful to remember on which version it is built upon)
- a path to the dependency (can be a CDN or a local folder)
- script and stylesheet to respectively pass css and scripts

```
# handle dependency
card_css <- "bootstrap.min.css"
bs4_card_dep <- function() {
  htmlDependency(
    name = "bs4_card",
    version = "1.0",
    src = c(href = "https://maxcdn.bootstrapcdn.com/bootstrap/4.0.0/css/"),
    stylesheet = card_css
  )
}
```

We create the card tag and give it the bootstrap 4 dependency through the `attachDependencies()` function. In recent version of `htmltools`, we may simply use `tagList(tag, deps)` instead.

```
# create the card
my_card <- function(...) {
  cardTag <- withTags(
    div(
      class = "card border-success mb-3",
      div(class = "card-header bg-transparent border-success"),
      div(
        class = "card-body text-success",
        h3(class = "card-title", "title"),
        p(class = "card-text", ...)
      ),
      div(class = "card-footer bg-transparent border-success", "footer")
    )
  )

  # attach dependencies (old way)
  # htmltools::attachDependencies(cardTag, bs4_card_dep())

  # simpler way
  tagList(cardTag, bs4_card_dep())
}
```

```
}
```

We finally run our app:

```
# run shiny app
ui <- fluidPage(
  title = "Hello Shiny!",
  fluidRow(
    column(
      width = 6,
      align = "center",
      br(),
      my_card("blablabla. PouetPouet Pouet.")
    )
  )
)

shinyApp(ui, server = function(input, output) { })
```

With this approach, you could develop a package of custom dependencies that people could use when they need to add custom elements in shiny.

6.3 Another example: Importing HTML dependencies from other packages

You may know shinydashboard, a package to design dashboards with shiny. In the following, we would like to integrate the box component in a classic Shiny App (without the dashboard layout). However, if you try to include the Shinydashboard box tag, you will notice that nothing is displayed since Shiny does not have shinydashboard dependencies. Fortunately htmltools contains a function, namely `findDependencies` that looks for all dependencies attached to a tag. How about extracting shinydashboard dependencies? Before going further, let's define the basic skeleton of a shinydashboard:

```
shinyApp(
  ui = dashboardPage(
    dashboardHeader(),
    dashboardSidebar(),
    dashboardBody(),
    title = "Dashboard example"
  ),
  server = function(input, output) { }
)
```


6.3. ANOTHER EXAMPLE: IMPORTING HTML DEPENDENCIES FROM OTHER PACKAGES 57

We don't need to understand shinydashboard details. However, if you are interested to dig in, help yourself. What is important here is the main wrapper function `dashboardPage`. (You should already be familiar with `fluidPage`, another wrapper function). We apply `findDependencies` on `dashboardPage`.

```
deps <- findDependencies(  
  dashboardPage(  
    header = dashboardHeader(),  
    sidebar = dashboardSidebar(),  
    body = dashboardBody()  
  )  
)  
deps
```

`deps` is a list containing 4 dependencies:

- Font Awesome handles icons
- Bootstrap is the main HTML/CSS/JS template. Importantly, please note the version 3.3.7, whereas the current is 4.3.1
- AdminLTE is the dependency containing HTML/CSS/JS related to the admin template. It is closely linked to Bootstrap 3.
- shinydashboard, the CSS and javascript necessary for shinydashboard to work properly. In practice, integrating custom HTML templates to shiny does not usually work out of the box for many reasons (Explain why!) and some modifications are necessary.

```
[[1]]  
List of 10  
 $ name      : chr "font-awesome"  
 $ version   : chr "5.3.1"  
 $ src       :List of 1  
 ..$ file: chr "www/shared/fontawesome"  
 $ meta      : NULL  
 $ script    : NULL  
 $ stylesheet: chr [1:2] "css/all.min.css" "css/v4-shims.min.css"  
 $ head      : NULL  
 $ attachment: NULL  
 $ package   : chr "shiny"  
 $ all_files : logi TRUE  
 - attr(*, "class")= chr "html_dependency"  
[[2]]  
List of 10  
 $ name      : chr "bootstrap"  
 $ version   : chr "3.3.7"  
 $ src       :List of 2
```

```

..$ href: chr "shared/bootstrap"
..$ file: chr "/Library/Frameworks/R.framework/Versions/3.5/Resources/library/shiny/www"
$ meta      :List of 1
..$ viewport: chr "width=device-width, initial-scale=1"
$ script    : chr [1:3] "js/bootstrap.min.js" "shim/html5shiv.min.js" "shim/respond.min.js"
$ stylesheet: chr "css/bootstrap.min.css"
$ head      : NULL
$ attachment: NULL
$ package   : NULL
$ all_files : logi TRUE
- attr(*, "class")= chr "html_dependency"
[[3]]
List of 10
$ name      : chr "AdminLTE"
$ version   : chr "2.0.6"
$ src       :List of 1
..$ file: chr "/Library/Frameworks/R.framework/Versions/3.5/Resources/library/shinydashboard"
$ meta      : NULL
$ script    : chr "app.min.js"
$ stylesheet: chr [1:2] "AdminLTE.min.css" "_all-skins.min.css"
$ head      : NULL
$ attachment: NULL
$ package   : NULL
$ all_files : logi TRUE
- attr(*, "class")= chr "html_dependency"
[[4]]
List of 10
$ name      : chr "shinydashboard"
$ version   : chr "0.7.1"
$ src       :List of 1
..$ file: chr "/Library/Frameworks/R.framework/Versions/3.5/Resources/library/shinydashboard"
$ meta      : NULL
$ script    : chr "shinydashboard.min.js"
$ stylesheet: chr "shinydashboard.css"
$ head      : NULL
$ attachment: NULL
$ package   : NULL
$ all_files : logi TRUE
- attr(*, "class")= chr "html_dependency"

```

Below, we attach the dependencies to the `box` with `tagList`, as shown above. Notice that our custom `box` does not contain all parameters from `shinydashboard` but this is not what matters in this example.

```
my_box <- function(title, status) {  
  tagList(box(title = title, status = status), deps)  
}  
ui <- fluidPage(  
  titlePanel("Shiny with a box"),  
  my_box(title = "My box", status = "danger"),  
)  
server <- function(input, output) {}  
shinyApp(ui, server)
```

Now, you may imagine the possibilities are almost unlimited! Interestingly, this is the approach we use in `shinyWidgets` for the `useBs4Dash` function and other related tools.

6.4 Suppress dependencies

In rare cases, you may need to remove an existing dependency (conflict). The `suppressDependencies` function allows to perform that task. For instance, `shiny.semantic` built on top of `semantic ui` is not compatible with Bootstrap. Below, we remove the `AdminLTE` dependency from a `shinydashboard` page and nothing is displayed (as expected):

```
shinyApp(  
  ui = dashboardPage(  
    dashboardHeader(),  
    dashboardSidebar(),  
    dashboardBody(suppressDependencies("AdminLTE")),  
    title = "Dashboard example"  
  ),  
  server = function(input, output) { }  
)
```


Chapter 7

Other tools

7.1 CSS

- See `cascadess` to customize the style of tags

```
ui <- list(  
  cascadess(),  
  h4(  
    .style %>%  
      font(case = "upper") %>%  
      border(bottom = "red"),  
    "Etiam vel tortor sodales tellus ultricies commodo."  
  )  
)
```


Practice

In this chapter, you will learn how to build your own html templates taken from the web and package them, so that they can be re-used at any time by anybody.

Chapter 8

Template selection

There exists tons of HTML templates over the web. However, only a few part will be suitable for shiny, mainly because of what follows:

- shiny is built on top of Bootstrap 3 (HTML, CSS and Javascript framework), meaning that going for another framework might not be straightforward. However, shinymaterial and shiny.semantic are examples showing this can be possible.
- shiny relies on jQuery (currently v 3.4.1 for shiny). Consequently, all templates based upon React, Vue and other Javascript framework will not be natively supported. Again, there exist some examples for React with shiny and more generally, the reactR package developed by Kent Russell and Alan Dipert from RStudio.

See the github repository for more details about all dependencies related to the shiny package.

Notes: As shiny depends on Bootstrap 3.4.1, we recommend the user who would like to experiment Bootstrap 4 features to be particularly careful about potential incompatibilities. See a working example here with bs4Dash.

A good source of **open source** HTML templates is Colorlib and Creative Tim. You might also buy your template, but forget about the packaging option, which would be illegal in this particular case.

Chapter 9

Define dependencies

Chapter 10

Template skeleton

Chapter 11

Develop custom input widgets

11.1 How does Shiny handle inputs?

11.2 How to add new input to Shiny?

Chapter 12

Testing templates elements